2017

Building an 8-bit Adding Microprocessor from Fundamental Logic Gates

MARLEY SEXTON

The most basic element of a computer is known as a *bit*. A bit is a kind of electronic "switch" that can be either *on* or *off*. When we talk about the value of a bit, however, we typically refer to it as being either *1* or *0*, taken to mean *on* or *off* respectively, or alternately *true* or *false* when we are talking about performing logical operation. For example if we were to take two bits and ask the question *"do both bits have a value of 1*" the answer would be either *true* or *false*. In computing we could then store this answer in a third bit that would be either *on* or *off*. If we do this we then have to assign the value *0* or *1* to either *true* or *false* – since our question has only those two answers. Traditionally we would store *1* as *true* and *0* as *false*. Thus the value of the bit could be correctly identified as being *on* or *off*, *1* or *0*, *true* or *false*.

Regardless, the nuances should hopefully sort themselves out in time during this work. After all, we're here to build an adder. Unfortunately, before we get down to the nitty gritty electronics side of things we have to go through an ocean of basic theory. To compute with our computer we first need to get to grips with *logic gates*, and then use these logic gates to construct detailed little circuits that can add together two basic 8-bit numbers.

Logic Gates

Once we have constructed bits, which we'll get to at a later stage, we next need to find a way to manipulate them to do our bidding – whether that be functioning as a calculator, outputting data, or just simply looking for the answer to a complex mathematical question. To do all these things we will make use of one of the most fundamental aspects of digital computers: the logic gates – a small homogenous family that takes input and then outputs a result, which can then be used as input for another process, and so on. Traditionally logic gates take two inputs – in our uses two *bits*, and then outputs the result into another bit. For example, if we want a bit to be 1 if, and only if, a set of two different bits are 1 then we would make use of an AND gate. Many processors make use of tertiary logic gates, which take three inputs and output one result. For our purposes, we're going to keep it simple and make use of binary logic: two inputs that leads to one output.

So how many possible logic gates are there? Well this we can calculate. We know that bits can only have one of two values, and we know that if there are two bits there are four possible combinations of one and zero. These are 0 and 0, 0 and 1, 1 and 0, and 1 and 1. This leads us to four possible results. Now a logic gate takes all these combinations into consideration and allows us to determine a result from them. For example we may want our logic gate output the result of 1 for the combinations 1 and 1 *and* 0 and 0, but output the result 0 for the combinations 1 and 0 *and* 0 and 1 – as shown in the table below:

Input Bit A	*Input Bit B*	*Output Bit*
1	1	0
1	0	1
0	1	1
0	0	0

We can visualize easily from this that given that there are four output bits in one of two states that there are 4×2 permutation, i.e. eight, and thus eight different types of logic gates. Functionally there are only two massively important gates – the NOR and NAND gates – since these posses what's known as *functional completeness*. They are a sort of universal gate that *all* the other gates can be made of, so if you know how to, for example, create a NOR gate you can not only create an array of NOR gates to produce the same effect as a NAND gate, but you can create arrays of varying complexity to produce the same results as every single other gate, which we'll look at.

Getting down to business we'll start with a nice example: the NAND gate. To do this, we first must explore its brother the AND gate.

Logic gates can ultimately be easily explained using some basic English, as their functions are so deeply ingrained in the language that all we need do it take it from the practical realm into the abstract. For example, let us say that you walk into your local coffee shop and ask "Is the cappuccino hot *and* milky?". The answer to this question can *only* be yes or no, and it is *yes* – or TRUE, so to speak – if and only if both of the conditions, hot and milky, are met. In all other scenarios, the answer is *no* – or FALSE. If the question were put through an AND logic gate, it would result in the same result.

Let's take a more abstract look at what is known as an *AND* logic gate. Let's take two friendly neighbourhood *bits* which we shall call *Alpha* (α) and *Beta* (β). We know that these bits can only be one of two things: *on* or *off*, *1* or *0*, *true* or *false*. Now a logic gate fundamentally takes two inputs – in this case, α and β - and output a result *dependent on the state of the inputs* (α and β). We'll use another happy bit called *Gamma* (*γ*) in which to store the product of our function.

In an AND logic gate gamma asks a simple question: "*are both α and β on (or TRUE)?*" If the answer is *yes,* then gamma is set to be on, since the statement "are both α and β on" is TRUE. If the answer is *no,* then gamma is set to be off, since the statement "are both α and β on" is FALSE.

It's worth noting an intermediate state between the above where α is on, and β is off, and vice versa. Is gamma on for this? We know that he is not, because both α and β must be on for γ to be on – one does not suffice.

Hopefully, that wasn't too confusing, but perhaps a nice table will help clear up any quandaries. First let's produce a table demonstrating the different permutations of the input bits: α and β. Knowing that they each can only be in one of two states, on or off. We can then infer that there are four possible combinations of these two bits. These are respectively "on and on", "on and off", "off and on" and "off and off". These are represented in the table below using numerical notation. Remember: 1 is *on* (or TRUE) and 0 is *off* (or FALSE).

α	*β*
1	1
1	0
0	1
0	0

Now let us bring our output bit, γ, into the mix. We've already gone over the different results for the varied input combinations above but it is worth repeating to ensure they are firmly planted within your mind. Thus, what line of the above table do you suppose γ will be on for, given that his value is intrinsically linked to the values of Alpha and Beta, and is only on if and only if *both* α *and* β are on? If you said row one, where both α and β are on, then you are correct! But what about for the other rows? Well in row two and three only one of the two input bits are on and we, being picky buggers, want them to *both* be on. In these cases then γ is off and set to 0. Likewise in the last row neither of the conditions are met so γ is, again, set to 0. Lets add our results to our table.

α	*β*	*γ*
1	1	1
1	0	0
0	1	0
0	0	0

Voila! That is in its earnest a truth table for an AND logic gate. As mentioned before it simply says "If α and β are TRUE then γ is TRUE". Mathematically speaking we can make this statement more succinct and simply say "if α and β, then γ."

As you may have noted the primary element of the AND logic gate is that magically word "and". This is known as the *logical operator* and technically it is given the fancy name of *logical conjunctive*. To go real abstract we can represent "and" with the symbol "∧". Thus our statement becomes "If α ∧ β, then γ". We can even then reduce *material* implication, the "if… then…" statement, to a short and sweet symbol: "⇒". Thus our statement becomes "α ∧ β ⇒ γ", which reads as above. Let's update our table with this in mind to aid in apprehension.

α	*β*	*α∧β*
1	1	1
1	0	0
0	1	0
0	0	0

In electronic diagrams, which we'll be making extensive use of later, this is represented with the symbol shown below. The logic traditionally flows from left to right, through the gate. The inputs and output are labelled appropriately.

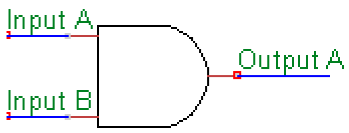


Figure 1 - 2 Input AND Gate

With this in mind let's move onto the star of the show, the NAND gate – something which was used extensively in early computers and can be found by the millions in modern Solid-State Hard Drive, among other components.

So what is a NAND gate? Put simply it is a *not* AND gate; it is the antithesis of our well understood AND gate. However, let's first take a step back and examine what a *not* function consists of. Known as *negation,* it is represented by that mysterious symbol at the upper left corner of most keyboards, next to the "1" key. I, of course, speak of the "¬" key, which is accompanied by its buddies "`" and "|", which for our purposes are irrelevant.

Negation is an inversion function. If we bring our little switches into the matter when we say "not *on*" we, of course, mean off, and when we say "not *off*" we naturally mean on. Easy to understand a truth table for a bit, α, and its negation is nonetheless shown below above its electronic diagram representation.

α	*¬α*
0	1
1	0

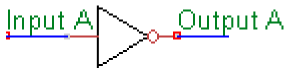


Figure 2 - A NOT Gate (Inverter)

Given that an inverter almost creates the "opposite" of a function, what do we suppose will be the result for a not AND gate? If we look at the truth table for an AND gate we note that it is TRUE for one result, and FALSE for three, thus when we negate this we instead realise that it is TRUE for three results and FALSE for one – since it reverses the result. Mathematically this is represented as "α ⊼ β" – as shown in the truth table below next to "α ∧ β".

α	*β*	*α∧β*	*α⊼β*
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	1

The electronic symbol for the NAND gate is shown below.

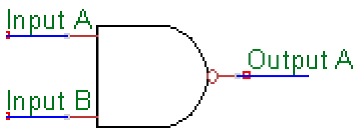


Figure 3 - 2 Input NAND Gate

If you remember back to earlier I mentioned that NAND gates are functionally complete and can be used to create all of the other gates. So in the spirit of exploration let's examine the AND gate and form it from an array of NAND gates. We know what result we are expecting and the simple solution to the problem is to use an inverter and negate the result of the NAND gates (a NAND gate if you will). However, since we're unable to use an inverter, since it's isn't one of our magical NAND gates, we'll have to find another way.

So let’s take our two inputs α and β and look to get our desired result into γ. First let α and β both be set to *1*. With the NAND gate this returns a result of 0 into γ. We know that with a NAND gate two inputs which are both set to 0 provides use with an output of 1, as per our truth table. Therefore, we can conclude that if we feed the results of *two* NAND gates into a third NAND gate then we will see our most converted “1” pop out the other side. To put more succinctly an AND gate with two inputs α and β can be expressed in terms of NAND gates thusly: (α ⊼ β) ⊼ (α ⊼ β). We can check this result by using different values for α and β other then 1. For example, if α or β is 1 whilst its counterpart is 0, or when both are set to 0, then α ⊼ β returns 1. Feeding two of these into another NAND gate we get an output of 0 – just what we’re looking for. An electronic schematic for this is shown below.

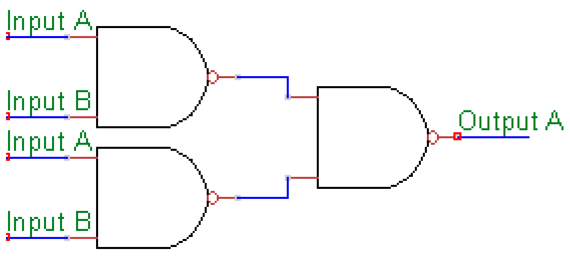


Figure 4 - AND gate constructed from NAND gates

Moving on, next I suppose we should probably examine the *other* functionally complete gate: the NOR gate. As you can likely allude from the *N* prefix, this is another *not-something* gate, in this case not-OR. One of the simpler gates to grasp an OR gate simple asks “are α *or* β TRUE?” As is evident, if α is 1, then the output is automatically 1, regardless of the state of β. In the same vain if β is TRUE, then the output is automatically 1, regardless of the state of α. The unique element of this gate is that if *both* α and β are TRUE, then the output is still TRUE, since in a roundabout way one or the other’s is on, regardless of the technicalities.

In mathematics, this function is known as a *logical disjunction* and is represented by the “V” symbol. The truth table for α V β is shown below next to the results of AND and NAND gates.

α	*β*	*α∧β*	*α⊼β*	*α∨β*
1	1	1	0	1
1	0	0	1	1
0	1	0	1	1
0	0	0	1	0

This is represented electronically as follows:

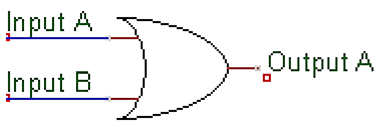


Figure 5 - 2 Input OR gate

Knowing what the NOT function did with the AND gate it is easy to understand what happens with the not-OR gate (NOR). Thus to same you the rigmarole associated with attempting to explain it in unnecessary amounts of detail the truth table for the NOR function, represented by the “⊽” symbol, is dictated below, next to all the other gates we have examined.

α	*β*	*α∧β*	*α⊼β*	*α∨β*	*α⊽β*
1	1	1	0	1	0
1	0	0	1	1	0
0	1	0	1	1	0
0	0	0	1	0	1

This is drawn in electrical diagrams as follows:

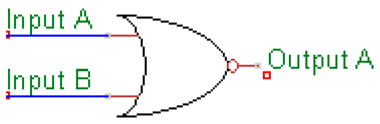


Figure 6 - 2 Input NOR gate

As mentioned previously NOR gates are, like NAND gates, functionally complete, and can be used to create all other gates. Knowing this it is worth spending some time creating their counterparts. The reasoning behind this is that if we can create a NAND gate out of NOR gates, and a NOR gate out of NAND gates then any other gate we create using any one of these functions can easily be transmuted into the other. For example, since we already know how to make an AND gate out of NAND gates if we can make NAND gates out of NOR gates then by the simple process of replacement, we can create an AND gate out of NOR gates.

Let's start easy and construct an OR gate out of NOR gates. Like the NAND gates we simply wish to invert an NOR gate to produce an OR gate but since this is out of the question we can at least garner some knowledge from our NAND AND gate example – that is that feeding the results of two identical logic gates with identical inputs into a third logic gate of the same make and model results in a negation of the said logic gate. That is to say that $(\alpha \bar{\vee} \beta) \bar{\vee} (\alpha \bar{\vee} \beta)$ is functionally an OR gate. Simple enough but I would like to elaborate and say that since we know that the if we have two inputs α and β both set to 0 then the result we want out the other end is also 0 in the case of our desired OR gate. We know however that in this scenario a NOR gate returns the result of 1. We *also* know that in every case where one input into a NOR is 1 it always returns FALSE. Thus by feeding the results of two NOR gates both with inputs of α and β into a third NOR gate we end up with a 0 popping out the other end. This is drawn electronically below:

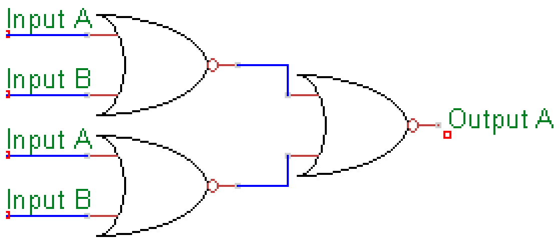


Figure 7 - OR gate constructed from NOR gates

It's also worth noting that since NOR gates function exactly like OR gates in that they only really depend on one of the inputs being TRUE you can create a OR gate from NOR gates by simply using the result of one NOR gate as *both* the inputs of a second NOR gate, since the only scenario in which it would output anything other then 0 is when the initial gate has two FALSE inputs. This is drawn below for reference.

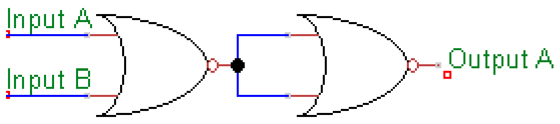


Figure 8 - Alternate construction of a OR gate from NOR gates

With this in place lets now create an NOR gate from NAND gates – we can also for ease of reference utilize AND gates since we can already form these from NAND constituents. Either way we know from our studies that a NOR gate only returns TRUE if and only if there are no TRUE inputs. So what we need is a way to see if either of the inputs are TRUE. One way that we can do this is feed each input, α and β, into it's *own* NAND gate, so that it returns a value of 0 if the input is 1 and 1 if the input is 0. For example if we do $\alpha \bar{\wedge} \alpha$ then if α is 1 the result is 0, and if α is 0 then the result is 1. If we then do this for *both* inputs then we can quickly conclude that if one of the outputs from either gate is 0 then one of the inputs, at the very least, is 1. Since one of the inputs would be 1 we can then automatically deduce that the overall NOR gate should return 0.

So let's backtrack and first work out how to perform a check to see if either of the outputs from the first gates are 1. Knowing that we can only use NAND gates the first logical things to do is to feed them into another NAND gate. Thus let's assume that all is well and after feeding each bit into it's own NAND gate out pops two values, for our example these will both be 0 – just what we want since it indicates that neither input is 1. Feeding both these values into a third NAND gate we get the result 0 ($1 \barwedge 1$). If either of the inputs into this third NAND gate were 0 it would naturally return a result of 0. Although this is what we ideally want it is simply solved by using the result of this third gate as both the inputs to a fourth NAND gate, which in essences acts as an inverter as we saw earlier. If the last inputs are both 1 it returns 0 and if both inputs are 1 it returns 0, as per the NAND logic table. Followed? Me neither, but it works and that is what ultimately matters. To make it easier to visualize here's how it would look in an electronic schematic:

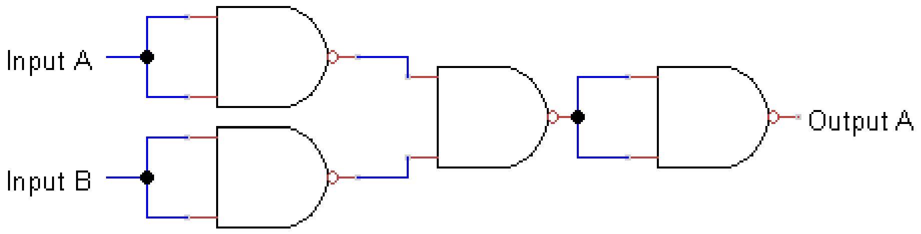


Figure 9 - NOR gate constructed from NAND gates

Okay, so now let's take a look at the OR gate made from NAND gates. We have already worked out how to make an OR gate from NOR gates as seen in the pictures above, so we need simply to replace the NOR gates in that example with the NOR gate made from NAND gates that we have just created. This looks schematically as below.

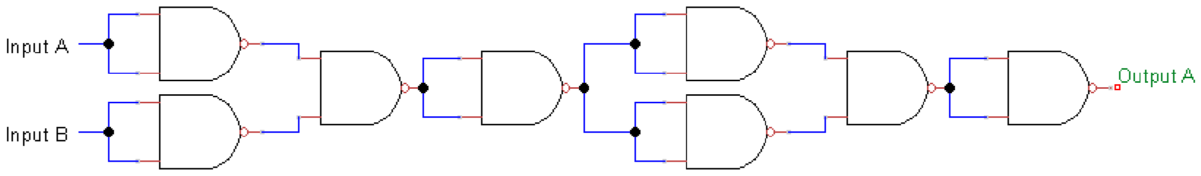


Figure 10 - OR gate constructed from 8 NAND gates

Or, alternatively as follows:

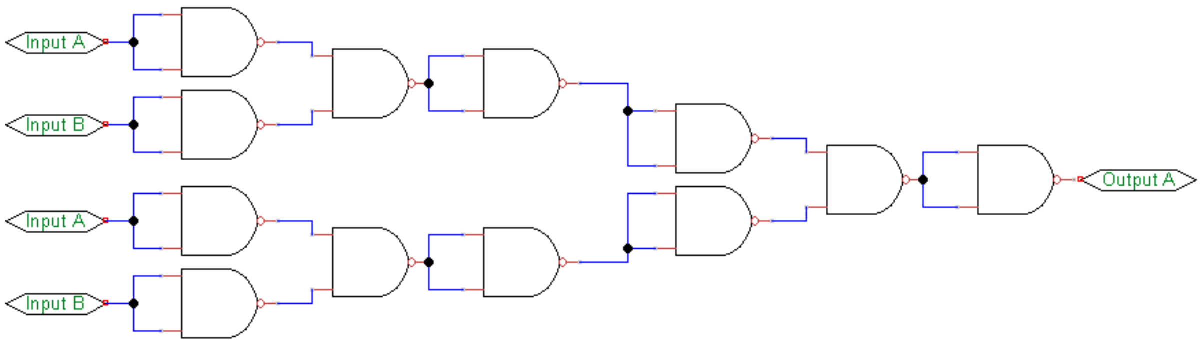


Figure 11 - OR gate constructed from 12 NAND gates

If that looks awfully complicated you're not wrong. Luckily there is an easier way to construct an OR gate using a far nicer number of NAND gates: three. If you remember back to when we constructed the NOR gate you may remember that the function of the fourth and last gate was to invert the result of of the previous three functions. Of course if we simply remove this inverter we then invert the NOR gate and bingo! We have much more compact OR gate, as shown below.

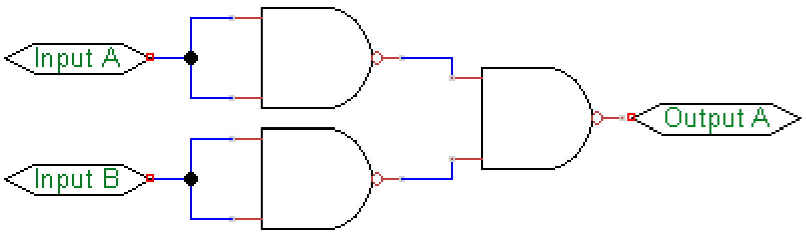


Figure 12 - OR gate constructed from 3 NAND gates

Lastly to make our circle complete we need to create a NAND gate from NOR gates. Once this is done we'll focus primarily on using NAND gates, but for the sake of functional completeness, it is worth noting since once we have it written in stone, we can do anything we'd like.

Thus we know from our truth table above that a NAND gate returns FALSE if and only if both inputs are TRUE, and FALSE in all other circumstances. Likewise, a NOR gate returns TRUE only when both inputs are FALSE, and TRUE in all other scenarios. Sound familiar? Just like the NOR gate constructed out of NAND gates, only reversed, we simply need to check if either of the inputs is TRUE, since if this is the case, then the NAND gate itself should return FALSE.

Luckily for us it requires an identical structure to the NOR gate made from NAND gates. We simply feed both inputs into their own NOR gates, and then feed these two results into a third NOR gate, before pushing that value through a fourth and final NOR gate. I'll save you the detailed explanation since it acts almost identically to the previous example only with the roles reversed – we want two TRUE values to pop out of the first two gates, for example, instead of two FALSE values. Either way the schematic for this is shown below:

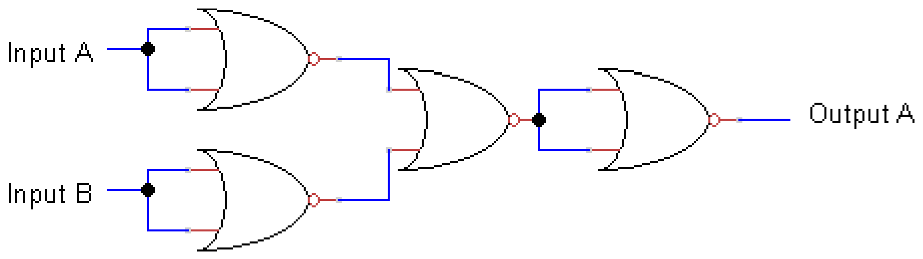


Figure 13 - NAND gate constructed from 3 NOR gates

Now we know this we can henceforth construct anything we can make out of NAND gates with NOR gates, and anything we can construct out of NOR gates with NAND gates – which makes our live simpler when we go on to actually get down to building an adder, since we'll primarily be using these two easy to construct logic gates. For now though it is only a reference, and for the sake of simplicity and space we will be using the more succinct forms. i.e. if we wish to use a OR gate we'll draw an OR gate as opposed to a huge array of NAND gates that will take up a tree's worth of paper space.

There still are still a couple of logic gates to cover, and we'll go over them briefly starting with the first in an exclusive family: the *XOR* gate. Known in logical circles as an *exclusive disjunction*, or as I prefer an *exclusive-or*, this gate is simple an OR gate with a slight twist. For two inputs α and β it is similar to the aforementioned OR gate in regards to that it is TRUE if α *or* β are TRUE, but unlike the OR gate a XOR gate is *FALSE* when both α *and* β are TRUE – it is said to *exclude* said result. In a way a XOR gate acts as you would intuitively expect an OR gate to work. Regardless the truth table for a XOR gate is shown below, using the "⊕" symbol – although "⊻" is equally correct but less distinctive.

α	β	$\alpha \wedge \beta$	$\alpha \barwedge \beta$	$\alpha \vee \beta$	$\alpha \overline{\vee} \beta$	$\alpha \oplus \beta$
1	1	1	0	1	0	0
1	0	0	1	1	0	1
0	1	0	1	1	0	1
0	0	0	1	0	1	0

This is drawn schematically as follows:

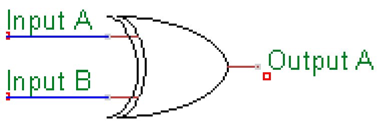


Figure 14 - 2 Input XOR gate

As is a running theme with our gates there does exist a *not*-XOR gate.; a XNOR gate. Like previous examples this complements the XOR gate and is equivalent to following a XOR gate with an inverter. It is the opposite, so to speak. As you may be able to garner from this vague description a XNOR gate practically checks to see if two given inputs are equal – it is TRUE if both inputs are either TRUE or FALSE, and is FALSE in cases where only one input is TRUE. As we will see later it is a monumentally useful gate that allows us to do some basic functions. Represented mathematically using the structurally similar "⊙" symbol, although "⊻̄" is equally correct, the truth table is shown below.

α	β	$\alpha\wedge\beta$	$\alpha\barwedge\beta$	$\alpha\vee\beta$	$\alpha\overline{\vee}\beta$	$\alpha\oplus\beta$	$\alpha\odot\beta$
1	1	1	0	1	0	0	1
1	0	0	1	1	0	1	0
0	1	0	1	1	0	1	0
0	0	0	1	0	1	0	1

Electronically this is drawn like so:

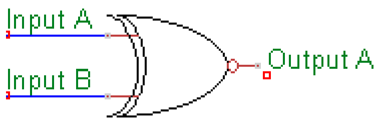


Figure 15 - 2 Input XNOR gate

That finally rounds up our list of fundamental logic gates. All that is needed to now construct our XOR and XNOR gates from a universal gate; either NOR or NAND. Luckily this time around we have four gates at our disposal: OR, NOR, AND and NAND since we know how to make each of these from our functionally complete gates.

Let's begin then with the XOR gate since it is the most closely related to one of our other gates, in this case, the OR gate. We know that our OR gate is correct for two inputs α and β for all possible states except one: when both are TRUE. So let's work with that situation first and attempt to correct it.

Thus if two inputs are TRUE and we wish to perform a function on it that produces a result of FALSE, then there are two gates we can look to: the NAND gate and the NOR gate. As previously mentioned, for all the other inputs an OR gate is more than suffice, so it stands to reason that by combining the results from an OR gate with the results of an AND gate we might go some way to solving our problem. Why is this? Well a NAND gate returns a single homogenous value for the other results. To help isolate our working out I have created a small truth table below demonstrating the results for a NAND gate and OR gate alone, with two inputs of α and β.

α	β	$\alpha\barwedge\beta$	$\alpha\vee\beta$	$\alpha\oplus\beta$
1	1	0	1	0
1	0	1	1	1
0	1	1	1	1
0	0	1	0	0

As we can see in our example only when both the NAND and OR gates return a value of TRUE should the XOR gate also return a value of TRUE. In all the other situations it should return a value of FALSE. That is to say that we should feed the results of these two gates into a third gate that only returns a value of TRUE when both inputs are TRUE. We of course know of a gate that displays this behaviour, the AND gate! Voila! We know have an XOR gate created from a Frankenstein mix of an OR gate, a NAND gate and an AND gate. Schematically this is drawn below:

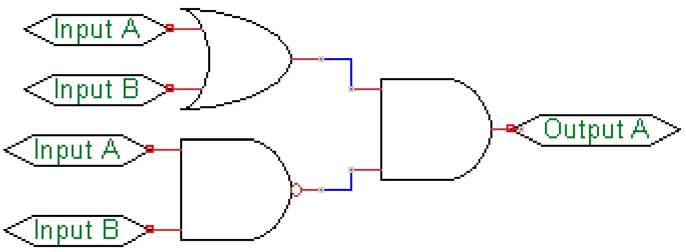


Figure 16 - XOR gate constructed from a OR gate, an AND gate and a NAND gate

With that in place we can replace each of the above gates with our NAND gate construction made earlier. When we do this is looks likes the monstrosity shown below.

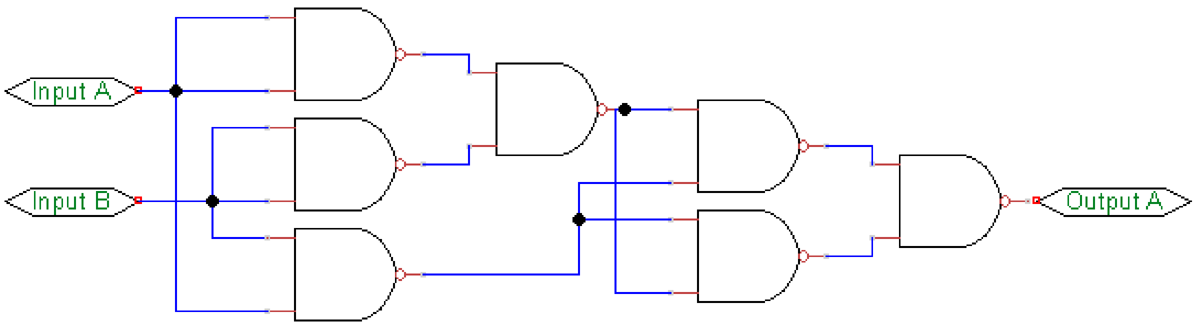


Figure 17 - XOR gate constructed from NAND gates

This again is complicated but not the simplest solution to our quandary. One of the benefits of using one set of gates only is that it is easy to visualize where we can cut corners. First let's look at the upper left quadrant, where there is two NAND gates fed by only one input each. If we remember to earlier this essentially checks whether the given input is TRUE or FALSE. It returns TRUE if the input is FALSE and FALSE if the input is TRUE. By feeding these two results into a third NAND gate we are essentially asking the question "Are the inputs identical?" – if so, it returns FALSE and if not it returns TRUE. This is of course what an OR gate does. However the left bottommost NAND gate answers the same question, in essence. It returns TRUE if the inputs are FALSE and FALSE if the inputs are TRUE. The only difference is that it can't differentiate between mixed inputs. If only one of the inputs is TRUE then it will still return TRUE.

We can now condense these gates down into a single NAND gate since the output for either of them for the "outermost" inputs – where both inputs are identical – is the same, but importantly the output for the middle inputs – where the inputs are mixed, i.e. 0 and 1 or 1 and 0 – is what we desire. The rightmost collection of NAND gates, which construct an AND gate, as we have seen before checks whether both inputs are identical. In our case, if we feed the two inputs into a NAND gate and then along with each input individually feed the result of said NAND gate into a third NAND gate then what we are checking is complicated but logically sound. In essence, we are checking to see if the inputs individually match the result of a NAND operation.

Perhaps it would be less confusing to walk through an example. Lets take two inputs that are both 1. If we use a NAND gate operation on both of these inputs together it will return a value of FALSE. If we use a NAND operation of this result together with each original input individual we will get two results which are both TRUE – since the inputs differ and aren't both TRUE. We want the ultimate result of this collection for the aforementioned input to be FALSE in accordance with what a XOR gate does, thus if we feed these two TRUE values into another NAND gate it will spit out a result of FALSE. Perfect! But what about with say two differing inputs of 1 and 0 or 0 and 1. For a XOR gate this should result in a TRUE value. Well, if we feed these inputs together into a NAND gate it will give us a result of TRUE once more. Checking this against the original inputs we will have one gate which takes two TRUE inputs, and one in which they differ. This will provide two differing results: TRUE and FALSE which when pushed into a final NAND gate will, since they are dissimilar, provide a result of TRUE. Again, just what we want. Finally lets take the last possible input where both inputs are FALSE. The first NAND gate we pop these into will return a value of TRUE. When we feed this result and each original input individually into a third and fourth gate since the inputs differ – one TRUE and one FALSE they *both* return TRUE. As we know when a NAND gate takes two TRUE inputs it returns FALSE, just what we want!

So that's that, hopefully not too perplexing but as with most things in logic, the best way to find the answer is to methodologically work through each example with a possible explanation. Anyhow the electronic schematic for this function, our XOR gate made from NAND gates, is shown below.

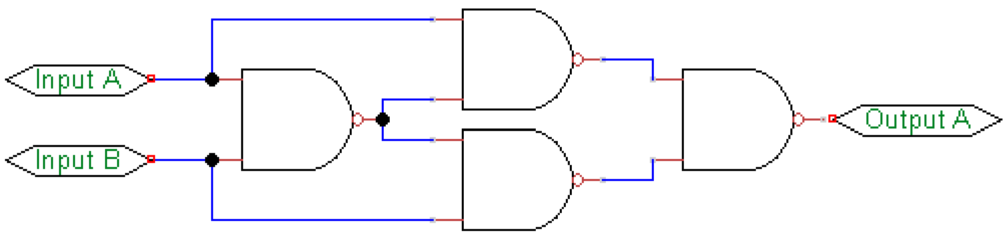


Figure 18 - XOR gate constructed from 4 NAND gates

Finally we can get around constructing a XNOR gate. Thankfully there is not a massively complicated rigmarole we have to go through this time since we already know how to invert a function using a NAND gate. Thus all we need to do is use the output of a XOR gate as both the inputs of an NAND gate – essentially strapping one on the end. This looks like as follows.

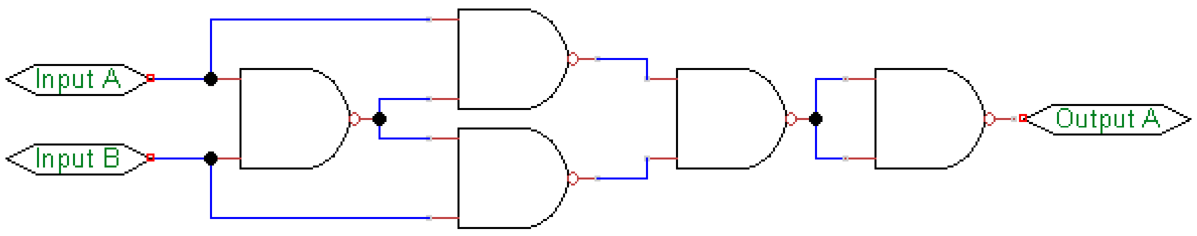


Figure 19 - XNOR gate constructed from 5 NAND gates

With that complete we have finished our roundup of the six basic logic gates (seven if you include the inverter). With these we'll be able to create any logical operation we wish to use, so let's get started.

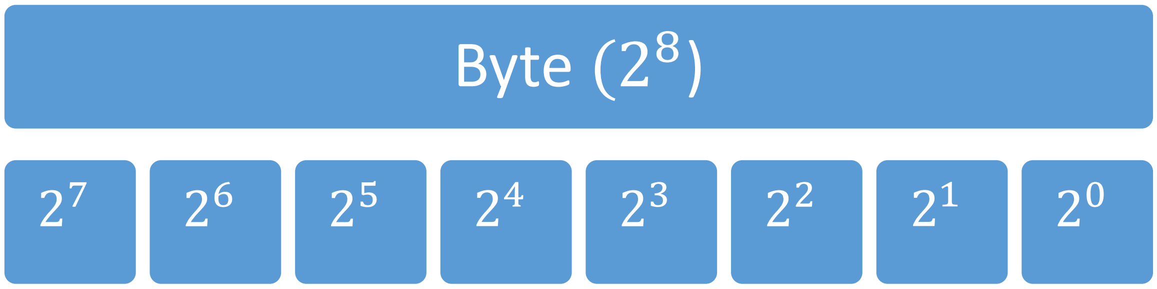
Bytes

Ah, the humble Byte. Simple and elegant it is one of the most widely used terms in computing. But what is a byte? Well, do you remember those bits we talked about? A byte is simply eight of them. For example, if we take eight bits each with a value of 1, then we can write the byte as “111111111”.

Why are bytes so important though? Well when we combine bits we typically assign each one of them a value. Just like the number “1093” has four figures – or values – a byte has eight figures. Since they can only be in one of two states, however, each bit, -working from right to left like in the traditional Arabic number system – is assigned a value on the order of 2^n $(2 \times 2 \times 2 \times \ldots)$ where n is the bit’s place in sequence. It is important to note that the rightmost digit of any binary number, not just a byte, is considers the “zeroth” bit. It is the *least significant* bit in sequence. For example, the third bit in the byte “00000100” represents the number 2^2, not 2^3, which is equal to 4 $(2^2 = 2 \times 2)$. To demonstrate this let’s count to five using *our* byte. The number “1” is represented by the bits “00000001” – i.e. 2^0. If we add “1” to this we note that the least significant bit *cannot* become “2” since we are is essence using a binary numbering system which allows digits to be one of two things: 1 or 0. Thus just like what happens when you go from 09 to 10 when counting normally, the least significant bit ”rolls over” and the next significant bit becomes a “1” and the least significant bit resets to “0”, thusly: “00000010”. If we increment this again the least significant bit can now tick upwards and we get “00000011”. What this says is $2^1 + 2^0$ which is equivalent to $2 + 1$ which obviously equals 3. Incrementing it once more to represent “4” we note that if the least significant bit rolls over to the next significant bit that too will roll over to the third bit. Thereofore we get “00000100” which tells us that it equals 2^2 which equals 4. Finally adding one once more we get the least significant bit incrementing so the byte becomes “00000101” which equals $2^2 + 2^0$ which is “5”. Hopefully you can see how this works now, and for reference the first 12 number in binary are shown below.

1	0001	5	0101	9	1001
2	0010	6	0110	10	1010
3	0011	7	0111	11	1011
4	0100	8	1000	12	1100

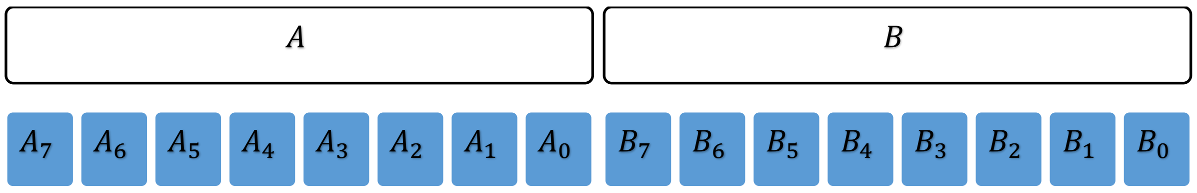
For reference a byte is broken down into individual bits as follows:



So how many numbers can our lovely byte represent? Well this we can calculate knowing what we learned above. There are 8 bits, each bit representing an exponentially larger number. This equates to $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$ which is equal to $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$, i.e. 255. If we include zero, this comes to 256. That’s not a lot of numbers but, like I said, the value of a bit doubles every time you move left. We’ll be building a simple 8-bit computer but it’s worth noting that modern computers are typically 32 bit, or – as is becoming more common – 64 bit. These can represent 4,294,967,295 and 18,446,744,073,709,551,616 numbers respectively. Although we could go for something like a 16 bit computer, which can represent 65 536 values, for simplicity we’ll stick with an 8 bit computer. That being said as you will see everything in this book can be “up-scaled” if you so desire. Let’s start easy then and create a small circuit to add two numbers together.

8 Bit Adder

So we want to add two bytes together. For this we need to do some setting up and formulate some notation that we can use to refer to individual bits as well as the bytes themselves. Thus let's create two bytes which we shall name A and B. Each byte has, of course, 8 bits contained therein. We shall refer to these using a *subscript* – a small number between the values of 0 and 7. The least significant (right-most) bit in a bit shall be bit "0" and the most significant (left-most) bit shall be bit "7". For example if we say that byte A is "00010000" then the bit that is *on* in that byte will be referred to as A. The complete breakdown of the two bytes A and S are shown in the hierarchy chart below.



With that it's worth now creating two control bytes to go alongside each of our operands above. The purpose of these control bytes is to have various elements of the numbers. For example, we can easily represent "8" as "0000 1000", but how do we go about representing "-8"? One possible way that makes life easy is to half the byte's value and set zero dead in the middle at what would otherwise represent "128". i.e. "1000 0000" is zero, "1000 0001" is "1" and "0111 1111" is "-1". This however easy it may be restricting our number range to 0 to 128, or rather negative 128 to positive 128. This is great but it isn't much to work with. By introducing a "signed" bit we can provide the attribute of "positive" or "negative" to each byte, doubling the volume of numbers we have to work with. Let's do this now.

So we're going to create two accompanying bytes to go with each of our operand bytes above. We'll use a superscript to refer to these. Thus we create two more bytes R^2 and S^2. The least significant bit of these, R_0^2 and R_0^2, will be the signed flag. We will use the other 7 bits in these bytes later, but for now they are irrelevant.

Next if we're going to add two bytes together we need another byte in which to store the value. Lets call this the *result* byte and refer to it as R. Again the bits contained therein are referred to with subscript notation. We also need to create him a buddy, R^2, in which to store information. Importantly however we'll be using an additional bit contained within R^2. Not only will R_0^2 be a signed flag but the next most significant bit, R_1^2 will be a *carry* flag. The purpose of this flag is to tell us when the product of the addition operation is greater than can be written in the byte itself. For example, if we add 16 to 255 the answer is 271. Since R only contains 8 bits this number cannot be stored therein, but we may need to use the information for a later calculation. Thus we increment the carry flag, R_1^2. This acts as an additional bit for our byte to use, and although R will roll over to "0000 1000", we still know that the answer is 271 and not 16. This comes in very handy for computing certain operations.

Either way moving on we now have a large array of bytes to use, these being R, R^2, A, A^2, B, and B^2. Keeping it simple however let's start with adding two positive numbers. But how do we do this? Well it's worth taking a look at traditional arithmetic. For the sake of argument lets say we want to add 123 to 191. How would be do this. Of course many of you will be able to perform the calculation in your head but if you wanted to write it down how would you do it? You'd write one number under the other, and then starting from the right – the least significant figure – you'd add them. If it went above 10 you'd carry the 1 and add it the next significant bit. Thus starting from the right, $1 + 3 = 4$. This doesn't roll over, since it doesn't exceed 10, so you would not carry anything. Next $2 + 9 = 11$. Here the value exceed 10, so we write a 1 under that column and then carry the 1. For the last digits we do $1 + 1$ and then include the carry. I.e. $1 + 1 + 1 = 3$. Writing this under the third column and we get the answer: 314.

	1	2	3
+	1	9	1
Result:	3	1	4
Carry:		1	

The same procedure is used on our bytes, except naturally they produce a carry far more often. Anyhow lets take our two least significant bits, A_0 andB_0 and create a *half adder* to perform addition on them such that $A_0 + B_0 = R_0 + C_0$, where C_0 is the intermediate carry – we'll only use $R_0^2$to store the carry once we have complete the whole calculate on the byte. Given that there are two possible values for both A_0 and B_0 – 1 or 0 – we can create a truth table to determine what should be the result when we add them. For example, if both A_0 and B_0 are 1 then adding them together the resultant bit R_0 is 0 - since 01 + 01 = 11 in binary - in addition to a carry which will be used in the addition of A_0 and B_0. If either A_0 or B_0 is 1, whilst it's counterpart is 0 then $R_0 = 1$, since $0 + 1 = 1$, and there is no carry. The result for $A_0 + B_0$ when both are 0 is obviously 0 with no carry. This is represented in the table truth table below.

A_0	B_0	R_0	C_0
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

You may recognize this truth table. The resultant bit, R_0, comes from pushing A_0 and B_0 through a XOR gate, whilst the carry bit, C_0, comes from using them as inputs in a AND gate. This is to say then that $A_0 \oplus B_0 = R_0$ and $A_0 \wedge B_0 = C_0$. A visual representation of this is shown below.

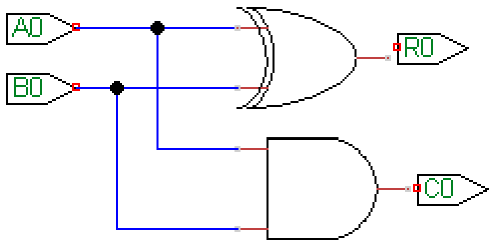


Figure 20 - Two Input Half-Adder

Now that we have constructed a half adder we can move onto constructing the star of the show: the full adder. Why do we need to do this? Well if we take a look at our previous example we note that it only takes two inputs – the two bits we wish to add – however with a full adder we need to take into consideration the carry, thus we require a three-input adder that outputs a result with any subsequent carry. So let's do this with our second most significant bits, A_1 and B_1, in addition to the previous carry C_0, such that $A_1 + B_1 + C_0 = R_1 + C_1$ – note that we are using a different carry bit. This bit will not be stored but instead it is used only in the addition operation.

This time we have double the number of possible combinations since we have three values to add. The complete range of possible permutations is shown in the table below.

A_1	B_1	C_0
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0

In the top line of the table, where A_1, B_1 and C_0 are all TRUE, we note that R_1 will equal “1” with a carry. This is because we add each of them in sequence. $1 + 1 = 10$, and then we increment it once more such that $1 + 1 + 1 = 11$. In all the rows where there are two values set to 1, $R_0 = 0$ with a carry, since $1 + 1 + 0 = 10$. Lastly the other rows are self explanatory. Any row with one “1” results in $R_0 = 1$ with no carry, and rows where there are no positive values have an R_0 value of 0 with no carry. Lets update our table.

A_1	B_1	C_0	R_1	C_1
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

This looks horrendous and the question now becomes how do we represent this logically. Since we can’t perform logical operations with three inputs we have to break it down into intermediate steps. For this we may as well start by adding A_1 and B_1. We know from earlier that the result of this is equal to $A_1 \oplus B_1$. We then need to add the carry to that. Let’s first add $A_1 \oplus B_1$ to the truth table to see where it leads us.

A_1	B_1	$A_1 \oplus B_1$	C_0	R_1	C_1
1	1	0	1	1	1
1	1	0	0	0	1
1	0	1	1	0	1
1	0	1	0	1	0
0	1	1	1	0	1
0	1	1	0	1	0
0	0	0	1	1	0
0	0	0	0	0	0

Carefully looking down the table you may notice a familiar pattern. Namely that there are only four combinations of $A_1 \oplus B_1$ and C_0 and they are all result in the same value of R_1. Let’s extract these.

$A_1 \oplus B_1$	C_0	R_1
1	1	0
1	0	1
0	1	1
0	0	0

Look familiar? It’s our friend XOR again. As perhaps expected to add $A_1 \oplus B_1$ and C_0 we simply use the result from $A_0 \oplus B_0$ along with C_0 as the inputs to a XOR gate. Therefore $R_1 = (A_1 \oplus B_1) \oplus C_0$, as demonstrated below.

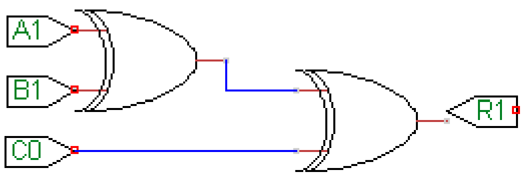


Figure 21 - Three input adder

Next, we have to calculate the second carry. If we spend some time looking through that large truth table above, we might some difficulty discovering any discernible pattern. We can note some a few things, however. Looking at the table, we can see that if A_1 and B_1 equal 1 then, regardless of the value of C_0, C_1 equals 1. On the other side of the spectrum when A_1 and B_1 equal 0 then, regardless of the value of C_0, C_1 equals 0. The difficulty arises in the intermediate situations however. In some rows C_1 is 0 when either one or the other of A_1 and B_1 is 0 and in some cases the opposite is true. We can see however that in the middle rows $A_1 \oplus B_1$ matches up with C_0 to create a nice pattern such that when both $A_1 \oplus B_1$ and C_0 are 1, C_1 is 1, and when they're both 0 C_1 is 0. This is to be expected since $A_1 \oplus B_1$ basically tells us that adding A_1 and B_1 does not result in a roll over – but that the sum is 1 - and in the cases where we add C_0 to this result it then naturally rollsover and produces a carry. Now, how do we put all of this into logical notation.

Well lets first work with $A_1 \oplus B_1$ and C_0. We've already discuss how if both $A_1 \oplus B_1$ *and* C_0 are 1 then C_1 is TRUE so it makes sense to and this AND function to our table in the form of $(A_1 \oplus B_1) \wedge C_0$.

A_1	B_1	$A_1 \oplus B_1$	C_0	$(A_1 \oplus B_1) \wedge C_0$	R_1	C_1
1	1	0	1	0	1	1
1	1	0	0	0	0	1
1	0	1	1	1	0	1
1	0	1	0	0	1	0
0	1	1	1	1	0	1
0	1	1	0	0	1	0
0	0	0	1	0	1	0
0	0	0	0	0	0	0

We can now see that $(A_1 \oplus B_1) \wedge C_0$ matches half the values of C_1 but not in those cases where both A_1 and B_1 are 1. We've already discussed how these appear to work on AND logic, since C_1 is TRUE if and only if both A_1 *and* B_1 are TRUE. So let now add another column to our already gargantuan table, that being $A_1 \wedge B_1$

A_1	B_1	$A_1 \oplus B_1$	$A_1 \wedge B_1$	C_0	$(A_1 \oplus B_1) \wedge C_0$	R_1	C_1
1	1	0	1	1	0	1	1
1	1	0	1	0	0	0	1
1	0	1	0	1	1	0	1
1	0	1	0	0	0	1	0
0	1	1	0	1	1	0	1
0	1	1	0	0	0	1	0
0	0	0	0	1	0	1	0
0	0	0	0	0	0	0	0

If we once again carefully look across we can now a nice correlation between the values of $A_1 \wedge B_1$ and $(A_1 \oplus B_1) \wedge C_0$ and the result we want, C_1 . We can see then when either $A_1 \wedge B_1$ *or* $(A_1 \oplus B_1) \wedge C_0$ is "1" then C_1 is 1, and we can also note then when both the aforementioned values are 0 then C_1 is equal to 0. One case doesn't arise however: both $A_1 \wedge B_1$ and $(A_1 \oplus B_1) \wedge C_0$ being 0. Luckily for us we needn't worry. We have every situation accounted for in the above table, with every possible permutation of inputs listed, and the case where both of our members of special interest are 0 together does not arise, and therefore *wont* arise – we've covered every eventuality. SO let's extract what we have learned into a table.

$A_1 \wedge B_1$	$(A_1 \oplus B_1) \wedge C_0$	C_1
1	1	?
1	0	1
0	1	1
0	0	0

We can see that the table potentially matches up with both OR and XOR gates, but which do we pick? Well since C_1 determine the carry that is added to the next bits of A and B it is worth playing it safe and using a XOR gate, since then we are certain that, whatever happens the process behaves in a meaningful way free of errors. By setting the top line to 0 we are basically taking out insurance against the unexpected. Let's combine all this then and use a XOR function to create the following statement:

$$C_1 = (A_1 \wedge B_1) \oplus ((A_1 \oplus B_1) \wedge C_0)$$

Putting that together with the function $(A_1 \oplus B_1) \oplus C_0$ which gives us the result bit, R_1, we get a nice schematic that looks as follows.

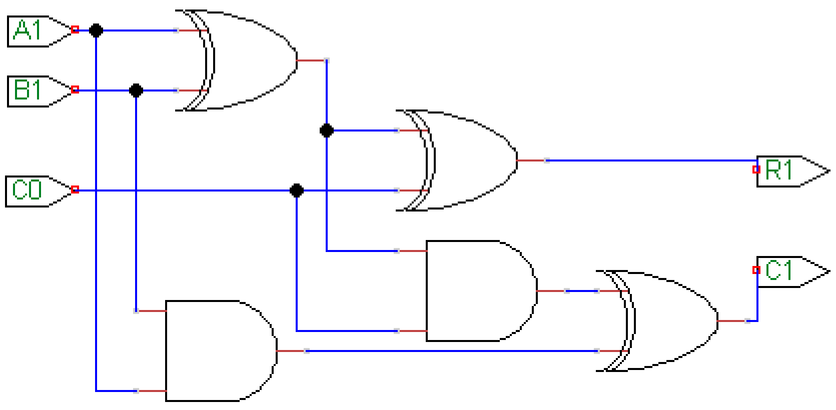
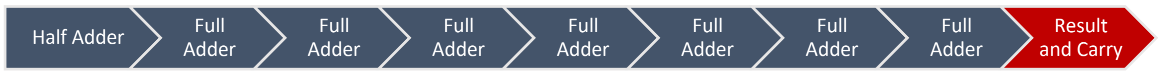


Figure 22 - Three Input Full Adder

To add any subsequent bits you simply feed the two bits you wish to add and the previous carry into the same locations as the previous bits and carry above. For example in the above diagram replacing A_1, B_1 and C_0 with A_2, B_2 and C_1 respectively, we'll get the results for R_2 and C_2. Continue until we have added all the 8 bits and feed the final carry (which would be C_7) into the companion byte to R_1, R^2 in bit 1 (R^2_1) and we'll have constructed and full 8 bit adder, as per the flow chart below.



With the same logic you can of course expand this onwards to 16, 32, 64 or even 128 bits if you're feeling adventurous. However, as you can see from the logic chart below, which demonstrates an 8 bit adder, it becomes extremely messy extremely fast.

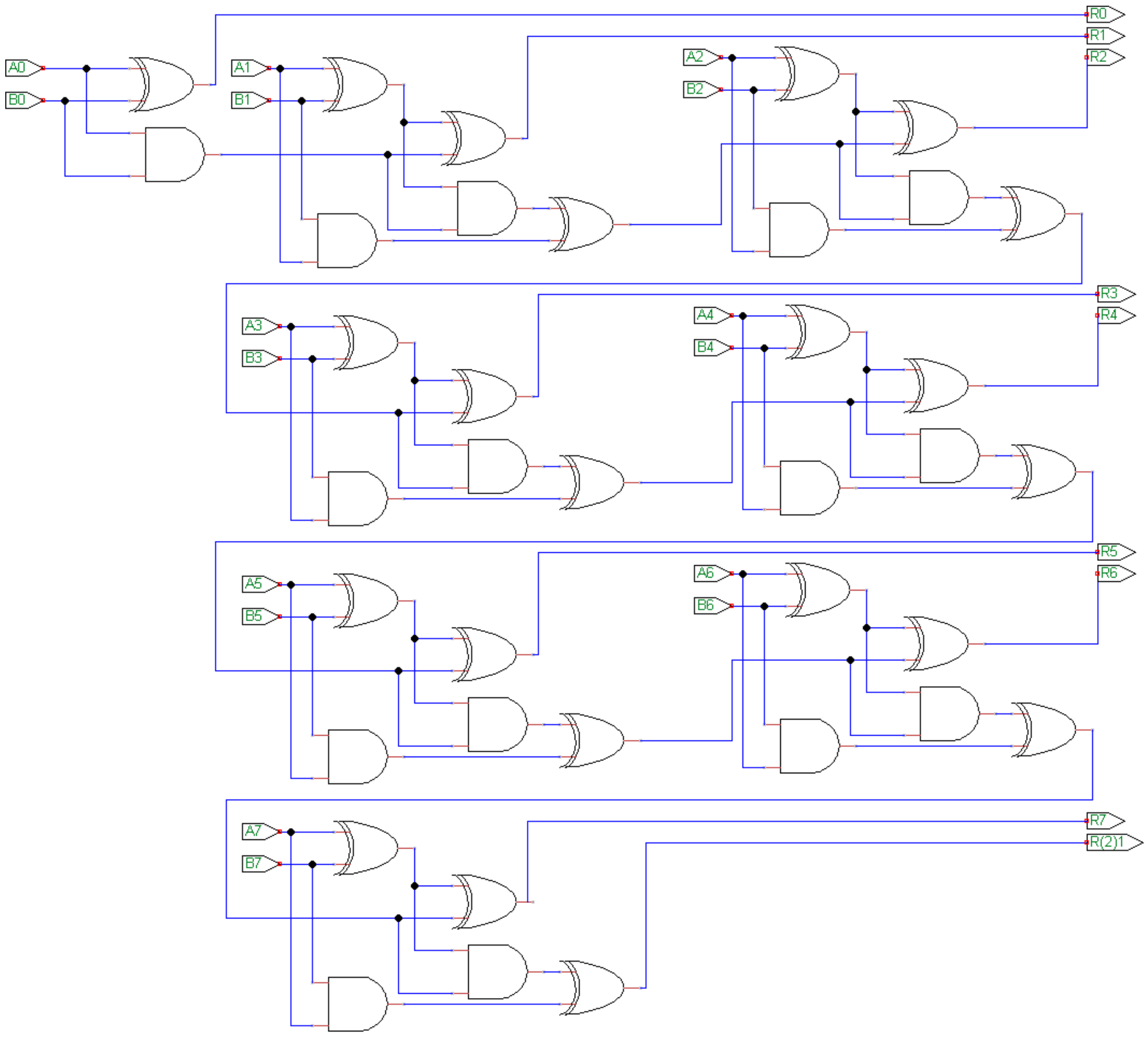


Figure 23 - 8 Bit Adder

For ease of reference, and so we can use it elsewhere in other components, we will be condensing components down into little microprocessors. Using a square symbol with basic input and output streams this will make it far, far easier to reference components without having to draw huge arrays of logic gates. For example, we'll represent the full 8-bit adder above with the following symbol, labelled with input and output nodes as appropriate.

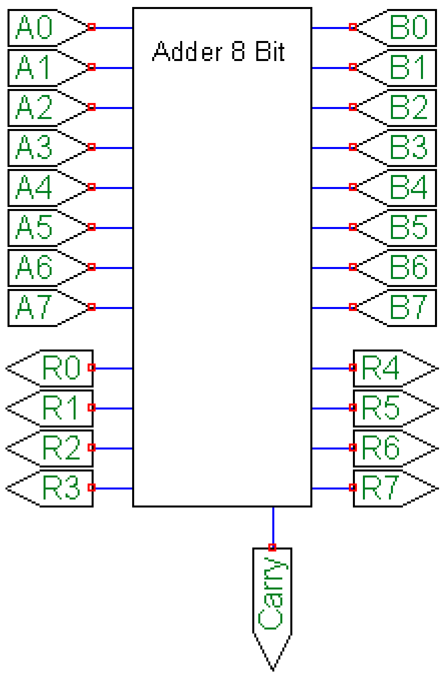


Figure 24 - Full 16 Input 8 Bit Adder

And there you have it: an 8-bit adder, as I mentioned earlier, it works perfectly fine when we want to add two positive numbers. If the sum of our two bytes exceeds the maximum a byte can theoretically store – which if you remember is 255 – then it will simply produce a ninth carry bit upon completion. This ninth bit allows numbers up to 512 to be stored, at least temporarily. For that you would require something a little more complicated. For our purposes though, the logic has been laid and may be continued on for as long as you need it too.